A Type-Theoretic Memory Model for Verification of Sequential Java Programs *

Joachim van den Berg, Marieke Huisman, Bart Jacobs, Erik Poll

Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
{joachim, marieke, bart, erikpoll}@cs.kun.nl

Abstract. This paper explains the details of the memory model underlying the verification of sequential Java programs in the "LOOP" project ([14, 20]). The building blocks of this memory are cells, which are untyped in the sense that they can store the contents of the fields of an arbitrary Java object. The main memory is modeled as three infinite series of such cells, one for storing instance variables on a heap, one for local variables and parameters on a stack, and and one for static (or class) variables. Verification on the basis of this memory model is illustrated both in PVS and in Isabelle/HOL, via several examples of Java programs, involving various subtleties of the language (wrt. memory storage).

1 Introduction

This paper reports on a (part of an) ambitious project to verify sequential Java programs, by making efficient use of modern tools for reasoning and translation. The underlying idea is that the quality of current proof tools (and hardware) should make it possible to change program verification from a purely theoretical discipline, working only for artificial, mathematically civilised programming languages, into a field where actual verification of programs written in a real-life language (namely Java [2, 6])—with all its messy semantical details—becomes feasible.

Here we isolate a small part of this larger project (see also [14, 20]) dealing with memory organisation and object creation (including initialisation). Therefore, many aspects are necessarily ignored, for example inheritance, casting, exception handling, and basic imperative programming. Some more information can be obtained from [14, 9, 8, 10], but many details are still unpublished. The initialisation procedure that we discuss here follows the Java language specification [6], and the examples we present do not involve certain ambiguities in the language specification about when to start static initialisation procedures, as pointed out in [4].

For our verification work we have developed a special purpose compiler, called LOOP, for *Logic of Object-Oriented Programming*. It is used as a front-end tool

* To appear in: *Workshop on Algebraic Development Techniques* (WADT), Springer LNCS 1827, 2000.

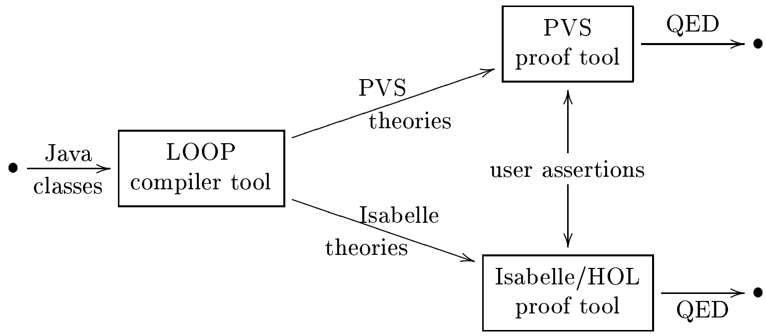

Fig. 1. Use of the LOOP tool, in combination with proof tools PVS or Isabelle/HOL

to a proof-tool, for which we can use both PVS [16] and Isabelle [17], see Figure 1, so that the tool is of interest for both prover communities. The LOOP tool turns Java classes[1] into logical theories, embodying the denotational semantics of these classes. A certain semantical prelude contains basic definitions, as explained in Section 5 below. A user can formulate his/her own assertions about the original Java class, and try to prove these on the basis of the logical theories that are produced. Typical are correctness assertions in pre-post-condition style, and class invariants. The semantics that is used is based on so-called coalgebras (see [13, 11] for some background information), but that is not really relevant in this paper. Coalgebras arise naturally in this setting since statements and expressions are state transformer functions with structured codomain types, see Section 5.

Of course, getting the semantics of Java right is the first prerequisite for any Java verification project. In this case, there are certain additional constraints.

1. It should be possible to formulate this semantics in the logic of the proof tool that is used. Since we use both PVS and Isabelle/HOL, this means that it should be formulated in typed higher order logic. Especially the typing constraints force us to use a tricky "untyped" definition of memory cells, because they should be capable of storing (the contents of the fields of) arbitrary objects. See Subsection 4.1 for details. In [5] there is also a memory model as basis for program verification in OBJ3. But this model avoids the problems that we are tackling here, because the programming language for which it is used is an imperative one without objects, and only has integer variables.
2. The type theoretic representation should allow efficient proofs, preferably using automatic rewriting. For example, at the lowest level, our translation involves various bureaucratic details about storage positions, but a user need not be concerned with these because appropriate rewriting lemmas take care of memory access.

[1] The LOOP tool is meant to be used only for Java classes which are accepted by a standard (JDK) compiler, and which are in particular type correct.

Thus the translation of Java classes is a fine balancing act. The current version is a result of many proof experiments, aimed at fully automatic memory handling by the back-end proof tool.

What we present is (part of) a *model-based* specification of a significant part of the Java programming language. It is not a *property-based* specification, largely for reasons of efficiency. What we present is a (type theoretic) memory model, and not an abstract property-based specification of a memory (like *e.g.* in [5, 18]). The memory locations in our memory model are natural numbers, simply because PVS and Isabelle are very efficient in handling natural numbers. This is relevant when, for instance a write operation at location n is followed by a read operation on location m. When $m \neq n$, this can be simplified to doing the read operation at m directly. The efficiency in the comparison of the locations n and m is lost in an abstract memory representation.

The three Java examples discussed in this paper form part of a larger series of examples [3]. These examples are used in lectures on Java, which do not aim at providing a systematic explanation of the semantics of Java, but of what is called *empirical semantics*: it consists of a large series of well-chosen, small examples of Java programs, each focusing on one specific aspect of the language[2]. This series of examples is a gold mine for the LOOP project: it allows us to test our semantics on a well-conceived set of examples coming from sources outside the LOOP project without knowledge about the particular semantic representation that we have chosen. About 30 of the examples have been translated into PVS and Isabelle with the LOOP tool and proven correct both with PVS and with Isabelle. Section 6 presents three representative (but small) examples from [3]. Correctness proofs are discussed in the subsequent Section 7.

This paper is organised as follows. It starts with a brief description of the higher order logic that will be used to explain our memory model. The primitive types and reference types of Java are translated to types in this logic. Section 4 explains the details of the memory model, and Section 5 describes the representation of Java statements and expressions that we use. Then, Section 6 introduces three short but non-trivial Java programs. The verification of two of these is discussed in Section 7.

2 Type-theoretic preliminaries

In this section we shall present the simple type theory and (classical) higher-order logic in which we will be working. It can be seen as a common abstraction from the type theories and logics of both PVS and Isabelle/HOL[3]. Using this general

[2] Together with a clear statement about the compiler version and machine that are used for producing certain significant results.

[3] Certain aspects of PVS and Isabelle/HOL are incompatible, like the type parameters in PVS versus type polymorphism in Isabelle/HOL, so that the type theory and logic that we use is not really in the intersection. But with some good will it should be clear how to translate the constructions that we present into the particular languages of these proof tools. See [7] for a detailed comparison.

type theory and logic means that we can stay away from the particulars of the languages of PVS and Isabelle and make this work more accessible to readers unfamiliar with these tools. Due to space restrictions, the explanation will have to be rather sketchy, and some experience in basic type theory is assumed on the reader's side.

Our type theory is a simple type theory with types built up from: type variables $\alpha, \beta, \ldots$, type constants nat, bool, string (and some more), exponent types $\sigma \to \tau$, labeled product (or record) types $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$ and labeled coproduct (or variant) types $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$, for given types $\sigma, \tau, \sigma_1, \ldots, \sigma_n$. New types can be introduced via definitions, as in:

– TYPE THEORY

$$\mathsf{lift}[\alpha] : \mathsf{TYPE} \stackrel{\text{def}}{=} \{\,\mathsf{bot} : \mathsf{unit} \mid \mathsf{up} : \alpha\,\}$$

where unit is the empty product type $[\,]$. This lift type constructor adds a bottom element to an arbitrary type, given as type variable α. It is frequently used in the sequel.

For exponent types we shall use the standard notation $\lambda x : \sigma.\, M$ for lambda abstraction and $N \cdot L$ for application. For terms $M_i : \sigma_i$, we have a labeled tuple $(\,\mathsf{lab}_1 = M_1, \ldots, \mathsf{lab}_n = M_n\,)$ inhabiting the corresponding labeled product type $[\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$. For a term $N : [\,\mathsf{lab}_1 : \sigma_1, \ldots, \mathsf{lab}_n : \sigma_n\,]$ in this product, we write $N.\mathsf{lab}_i$ for the selection term of type σ_i. Similarly, for a term $M : \sigma_i$ there is a labeled or tagged term $\mathsf{lab}_i\, M$ in the labeled coproduct type $\{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$. And for a term $N : \{\,\mathsf{lab}_1 : \sigma_1 \mid \ldots \mid \mathsf{lab}_n : \sigma_n\,\}$ in this coproduct type, together with n terms $L_i(x_i) : \tau$ containing a free variable $x_i : \sigma_i$ there is a case term $\mathsf{CASES}\ N\ \mathsf{OF}\ \{\,\mathsf{lab}_1\, x_1 \mapsto L_1(x_1) \mid \ldots \mid \mathsf{lab}_n\, x_n \mapsto L_n(x_n)\,\}$ of type τ which binds the x_i. These introduction and elimination terms for labeled products and coproducts are required to satisfy standard (β)- and (η)-conversions.

Formulas in higher-order logic are terms of type bool. We shall use the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\supset$ (implication), $\neg$ (negation, used with rules of classical logic) and constants true and false, together with the (typed) quantifiers $\forall x : \sigma.\, \varphi$ and $\exists x : \sigma.\, \varphi$, for a formula φ. There is also a conditional term $\mathsf{IF}\ \varphi\ \mathsf{THEN}\ M\ \mathsf{ELSE}\ N$, for terms M, N of the same type.

All these language constructs are present in both PVS and Isabelle/HOL.

3 Modeling Java's primitive types and reference types

The primitive types in Java are:

`byte`, `short`, `int`, `long`, `char`, `float`, `double`, `boolean`

The first five of these are the so-called integral types. They have definite ranges in Java (*e.g.* `int` ranges from -2147483648 to 2147483647). For all of these we shall assume corresponding type constants byte, short, int, long, char, float, double and bool in our type theory[4].

Reference types are used in Java for objects and arrays. A reference may be `null`, indicating that it does not refer to anything. In our model, a non-null reference contains a pointer 'objpos' to a memory location (on the heap, see Subsection 4.2), together with a string 'clname' indicating the run-time[5] type of the object, or the run-time elementtype of the array, at this location, and possibly two natural numbers describing the dimension and length of non-null array references. This leads to the following definition.

– TYPE THEORY

$$\begin{aligned} &\mathsf{RefType} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ &\quad \{\, \mathsf{null} \colon \mathsf{unit} \mid \mathsf{ref} \colon [\, \mathsf{objpos} \colon \mathsf{MemLoc}, \\ &\qquad\qquad\qquad\qquad \mathsf{clname} \colon \mathsf{string}, \\ &\qquad\qquad\qquad\qquad \mathsf{dimlen} \colon \mathsf{lift}[\,[\, \mathsf{dim} \colon \mathsf{nat}, \mathsf{len} \colon \mathsf{nat}\,]\,]\,]\,\} \end{aligned} \tag{1}$$

For reasons of abstraction, we use the type definitions:

– TYPE THEORY

$$\mathsf{MemLoc} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \mathsf{nat} \qquad \text{and similarly} \qquad \mathsf{CellLoc} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \mathsf{nat} \tag{2}$$

The latter type will be used below for locations in memory cells.

We have included the fields clname and dimlen in references. They could also be included in the memory cells that will be defined next. This does not really change the model.

4 The memory model

This section starts by defining memory cells for storing Java objects and arrays. They are used to build up the main memory for storing arbitrarily many such items. This object memory OM comes with various operations for reading and writing.

[4] One can take for example the type of integers $\ldots, -2, -1, 0, 1, 2, \ldots$ for the integral types, and the type of real numbers for the floating point types double and float, ignoring ranges and precision.

[5] Recall that a typical feature of object-oriented programming is that the static, declared type of a variable may be different from its dynamic, run-time type. The latter must be a subtype of the former, a property known as type-safety.

4.1 Memory cells

A single memory cell can store the contents of all the fields from a single object of an arbitrary class. The (translated) types that the fields of objects can have are limited to byte, short, int, long, char, float, double, bool and RefType. Therefore a cell will have entries for all of these. The number of fields for a particular type is not bounded, so we shall simply incorporate infinitely many in a memory cell:

– TYPE THEORY

$$\begin{array}{l} \mathsf{ObjectCell} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \big[\, \mathsf{bytes} \colon \mathsf{CellLoc} \to \mathsf{byte}, \\ \quad\;\; \mathsf{shorts} \colon \mathsf{CellLoc} \to \mathsf{short}, \\ \quad\;\; \mathsf{ints} \colon \mathsf{CellLoc} \to \mathsf{int}, \\ \quad\;\; \mathsf{longs} \colon \mathsf{CellLoc} \to \mathsf{long}, \\ \quad\;\; \mathsf{chars} \colon \mathsf{CellLoc} \to \mathsf{char}, \\ \quad\;\; \mathsf{floats} \colon \mathsf{CellLoc} \to \mathsf{float}, \\ \quad\;\; \mathsf{doubles} \colon \mathsf{CellLoc} \to \mathsf{double}, \\ \quad\;\; \mathsf{booleans} \colon \mathsf{CellLoc} \to \mathsf{bool}, \\ \quad\;\; \mathsf{refs} \colon \mathsf{CellLoc} \to \mathsf{RefType} \,\big] \end{array} \tag{3}$$

Recall that CellLoc is defined as nat in (2). Our memory will be organised in such a way that each memory location points to a memory cell, and each cell location to a position for a label inside the cell.

Storing an object from a class with, for instance, two integer fields and one Boolean field, in a memory cell is done by (only) using the first two values (at 0 and at 1) of the function $\mathsf{ints} \colon \mathsf{CellLoc} \to \mathsf{int}$ and (only) the first value (at 0) of the function $\mathsf{booleans} \colon \mathsf{CellLoc} \to \mathsf{bool}$. Other values of these and other functions in the object cell are irrelevant, and are not used for objects of this class. Enormous storage capacity is wasted in this manner, but that is unproblematic. The LOOP compiler attributes these cell positions to fields of a class, see Subsection 4.4.

Storing an array of Booleans, say of dimension 1 and length 100, in a memory cell is done by using the first 100 entries of the function $\mathsf{booleans} \colon \mathsf{CellLoc} \to \mathsf{bool}$, and nothing else. For arrays of objects the refs function is used. Similarly for multi-dimensional arrays.

An empty memory cell is defined with Java's default values (see [6, §§ 4.5.4]) for primitive types and reference types:

– TYPE THEORY

$$\mathsf{EmptyObjectCell} \colon \mathsf{ObjectCell} \stackrel{\text{def}}{=} \begin{array}[t]{l} (\,\mathsf{bytes} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{shorts} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{ints} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{longs} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{chars} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{floats} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{doubles} = \lambda n \colon \mathsf{CellLoc}.\, 0, \\ \ \mathsf{booleans} = \lambda n \colon \mathsf{CellLoc}.\, \mathsf{false}, \\ \ \mathsf{refs} = \lambda n \colon \mathsf{CellLoc}.\, \mathsf{null}\,) \end{array} \tag{4}$$

Storing an empty object cell at a particular memory position guarantees that all field values stored there get default values.

4.2 Object memory

Object cells form the main ingredient of a new type OM representing the main memory. It has a heap, stack and static part, for storing the contents of respectively instance variables, local variables and parameters, and static (also called class) variables:

– TYPE THEORY

$$\begin{array}{l} \mathsf{OM} : \mathsf{TYPE} \stackrel{\text{def}}{=} \\ \qquad [\,\mathsf{heapmem} \colon \mathsf{MemLoc} \to \mathsf{ObjectCell}, \\ \qquad\ \ \mathsf{heaptop} \colon \mathsf{MemLoc}, \\ \qquad\ \ \mathsf{stackmem} \colon \mathsf{MemLoc} \to \mathsf{ObjectCell}, \\ \qquad\ \ \mathsf{stacktop} \colon \mathsf{MemLoc}, \\ \qquad\ \ \mathsf{staticmem} \colon \mathsf{MemLoc} \to [\,\mathsf{initialised} \colon \mathsf{bool}, \mathsf{staticcell} \colon \mathsf{ObjectCell}\,]\,] \end{array} \tag{5}$$

The entry heaptop (resp. stacktop) indicates the next free (unused) memory location on the heap (resp. stack). These change during program execution. The LOOP compiler assigns locations (in the static memory) to classes with static fields. At such locations a Boolean initialised tells whether static initialisation has taken place for this class. One must keep track of this because static initialisation should not be performed more than once.

So far we have introduced memory cells for storing all the fields of objects and all the entries of arrays. This is actually not the full story, as it applies only to cells on the heap:

– a cell on the stack is used for storing the local variables (and possibly parameters[6]) that are used in a particular Java scope, see Subsection 4.5.

[6] and possibly also a special return variable for temporarily storing the value of an expression `e` in a return statement `return e`.

– a cell in the static part of the memory is used for storing the static fields of a class.

Despite these differences, the general ideas in the three different rôles for memory cells are the same: cell positions are assigned to variables by the LOOP compiler, and the values of these variables can be accessed and changed via get- and put-operations that will be described in the next subsection.

Notice that if we have a local (reference) variable, say `Object obj`, then `obj` will be linked to an entry of the refs function of a cell on the stack. If this entry is a non-null reference, it will point to an object on the heap. So references may be on the stack or in the static part of the memory, but objects are always on the heap.

We should still emphasise that since the heap, stack and static memories in OM are all infinite, a Java `OutOfMemoryError` will never arise in our model. Also, the model does not involve any garbage collection (which should be transparent anyway).

4.3 Reading and writing in the object memory

Accessing a specific value in an object memory $x\colon \mathsf{OM}$, either for reading or for writing, involves the following three ingredients: (1) an indication of which memory (heap, stack, static), (2) a memory location of a cell (in MemLoc), and (3) a cell location (in CellLoc) giving the offset in the cell. These ingredients are combined in the following variant type for memory addressing.

– TYPE THEORY

$$\begin{aligned}
&\mathsf{MemAdr} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\
&\qquad \{\, \mathsf{heap}\colon [\, \mathsf{ml}\colon \mathsf{MemLoc}, \mathsf{cl}\colon \mathsf{CellLoc}\,] \\
&\qquad \mid \mathsf{stack}\colon [\, \mathsf{ml}\colon \mathsf{MemLoc}, \mathsf{cl}\colon \mathsf{CellLoc}\,] \\
&\qquad \mid \mathsf{static}\colon [\, \mathsf{ml}\colon \mathsf{MemLoc}, \mathsf{cl}\colon \mathsf{CellLoc}\,]\,\}
\end{aligned} \tag{6}$$

For each type typ from the collection of types byte, short, int, long, char, float, double, bool and RefType occuring in object cells, as defined in (3), we have two operations:

– TYPE THEORY

$$\begin{array}{rl}
\mathsf{gettyp}(x, m)\colon \mathsf{typ} & \text{for } x\colon \mathsf{OM}, m\colon \mathsf{MemAdr} \\
\mathsf{puttyp}(x, m, u)\colon \mathsf{OM} & \text{for } x\colon \mathsf{OM}, m\colon \mathsf{MemAdr}, u\colon \mathsf{typ}
\end{array}$$

We shall describe these functions in detail only for typ = byte; the other cases are similar. Reading from the memory is easy: for $x\colon \mathsf{OM}, m\colon \mathsf{MemAdr}$,

— TYPE THEORY

$$\begin{aligned}\mathsf{getbyte}(x,m) \stackrel{\text{def}}{=} \mathsf{CASES}\ m\ \mathsf{OF}\ \{ \\ &\mid \mathsf{heap}\,\ell \mapsto ((x.\mathsf{heapmem}\cdot(\ell.\mathsf{ml})).\mathsf{bytes})\cdot(\ell.\mathsf{cl}) \\ &\mid \mathsf{stack}\,\ell \mapsto ((x.\mathsf{stackmem}\cdot(\ell.\mathsf{ml})).\mathsf{bytes})\cdot(\ell.\mathsf{cl}) \\ &\mid \mathsf{static}\,\ell \mapsto ((x.\mathsf{staticmem}\cdot(\ell.\mathsf{ml})).\mathsf{staticcell.bytes})\cdot(\ell.\mathsf{cl})\ \}\end{aligned}$$

The corresponding write-operation uses updates of records and also updates of functions, for which we write a combined 'WITH', which is hopefully self-explanatory: for $x\colon \mathsf{OM}$, $m\colon \mathsf{MemAdr}$ and $u\colon \mathsf{byte}$,

— TYPE THEORY

$$\begin{aligned}&\mathsf{putbyte}(x,m,u) \\ &\stackrel{\text{def}}{=} \mathsf{CASES}\ m\ \mathsf{OF}\ \{ \\ &\qquad \mid \mathsf{heap}\,\ell \mapsto x\ \mathsf{WITH}\ [\,((x.\mathsf{heapmem}\cdot(\ell.\mathsf{ml})).\mathsf{bytes})\cdot(\ell.\mathsf{cl}) := u\,] \\ &\qquad \mid \mathsf{stack}\,\ell \mapsto x\ \mathsf{WITH}\ [\,((x.\mathsf{stackmem}\cdot(\ell.\mathsf{ml})).\mathsf{bytes})\cdot(\ell.\mathsf{cl}) := u\,] \\ &\qquad \mid \mathsf{static}\,\ell \mapsto x\ \mathsf{WITH}\ [\,((x.\mathsf{staticmem}\cdot(\ell.\mathsf{ml})).\mathsf{staticcell.bytes})\cdot(\ell.\mathsf{cl}) := u\,]\,\}\end{aligned}$$

The various get- and put-functions (18 in total) satisfy obvious commutation equations, like:

— TYPE THEORY

$$\begin{aligned}\mathsf{getbyte}(\mathsf{putbyte}(x,m,u),n) &= \mathsf{IF}\ m=n\ \mathsf{THEN}\ u\ \mathsf{ELSE}\ \mathsf{getbyte}(x,n) \\ \mathsf{getbyte}(\mathsf{putshort}(x,m,v),n) &= \mathsf{getbyte}(x,n)\end{aligned}$$

Such equations (81 together) are used for auto-rewriting: the back-end proof-tool simplifies goals automatically whenever these equations can be applied.

4.4 Object storage

Consider a Java class `C` with fields as below.

— JAVA

```
class C {
  int j, k;
  static boolean t = true;
  B b;
  float[] f;
  ... }
```

The LOOP compiler will reserve a special static location, say $s\colon \mathsf{MemLoc}$, for `C`. Let $p\colon \mathsf{MemLoc}$ be an arbitrary location, intended as location of an object of class C. The LOOP compiler will then, for a given object memory $x\colon \mathsf{OM}$, arrange the instance fields of that object in the memory cell in the heap of x at p, and the static fields in the cell in the static memory of x at s. This is done according to the following table.

field	value in $x\colon \mathsf{OM}$
j	$\mathsf{getint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0))$
k	$\mathsf{getint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1))$
t	$\mathsf{getbool}(x, \mathsf{static}(\mathsf{ml} = s, \mathsf{cl} = 0))$
b	$\mathsf{getref}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0))$
f	$\mathsf{getref}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1))$

This involves:

- Storing the integer value of j at the first location of the infinite sequence ints of integers in the heap memory cell at p in x, *i.e.* at $((x.\mathsf{heapmem}) \cdot p).\mathsf{ints} \cdot 0$; it can then be read as $\mathsf{getint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0))\colon \mathsf{int}$; and it can be modified, say to value $a\colon \mathsf{int}$, by $\mathsf{putint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0), a)$.
- Storing the integer value of k at the second location $((x.\mathsf{heapmem}) \cdot p).\mathsf{ints} \cdot 1$; this value can be read via $\mathsf{getint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1))$ and modified via $\mathsf{putint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1), a)$.
- Storing the value of the *static* Boolean t at the first location of the sequence booleans of Booleans in the static memory at s in x, *i.e.* at $((x.\mathsf{staticmem}) \cdot s).\mathsf{staticcell}.\mathsf{booleans} \cdot 0$. It can be read as $\mathsf{getbool}(x, \mathsf{static}(\mathsf{ml} = s, \mathsf{cl} = 0))$, and be modified to value $a\colon \mathsf{bool}$ via $\mathsf{putbool}(x, \mathsf{static}(\mathsf{ml} = s, \mathsf{cl} = 0), a)$. When these get- and put-operations are called, static initialisation of C may have to be done. This is indicated by the Boolean $(x.\mathsf{staticmem} \cdot s).\mathsf{initialised}$, which is set to true in static initialisation.
- Storing the object reference of b at the first location of the sequence refs of references, *i.e.* at $((x.\mathsf{heapmem}) \cdot p).\mathsf{refs} \cdot 0$, which is of type RefType. Recall from (1) that an element of type RefType may be null or a reference containing a pointer (with label objpos) to another heap memory cell where an object of class B is stored. The reference to the B object may be read and updated via the getref and putref operations.
- Storing the array reference of f at the second location of the sequence refs of references, *i.e.* at $((x.\mathsf{heapmem}) \cdot p).\mathsf{refs} \cdot 1$. If it is not null, it refers to a heap memory cell where the array of floats is actually located. This array reference value may be read and modified via getref and putref. How to access the array at a particular index will be described in the next section (especially in Figure 2).

It should be clear that organising the values of the instance and class variables in such a manner involves a lot of bookkeeping. This may be a problem for humans, but since the translation of Java classes is done by a tool, this bureaucracy is not a burden. Also, since the proof obligations at this memory level can be handled by automatic rewriting, the user does not need to see these details in proofs.

An assignment statement `j = k` for the above fields j, k from class C will ultimately come down to a state transformer function $\mathsf{OM} \to \mathsf{OM}$ sending a state

x to $\mathsf{putint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 0), \mathsf{getint}(x, \mathsf{heap}(\mathsf{ml} = p, \mathsf{cl} = 1)))$. Thus it will result in a memory update function.

4.5 Local variables and parameters

The contents of local variables and parameters are stored at the stack, following the allocation ideas outlined in the previous subsection. That is, if for instance three local variables are declared of type `int` in a method body, then they are stored in the first three integer positions of the ints function of the cell in stackmem at the next free position stacktop. Upon entry of a method body stacktop is incremented by one, allocating one cell for all local variables and parameters (plus return variable) in the body, and upon leaving the body stacktop is decremented by one. For instance, the interpretation of a block statement (inside a method body)

$$[\![\, \texttt{\{int i = 5; ...\}} \,]\!]$$

is a function $\mathsf{OM} \to \mathsf{StatResult}$ (see Section 5) given on $x \colon \mathsf{OM}$ by

– TYPE THEORY

$$
\begin{array}{l}
(\,\mathsf{LET} \\
\quad \mathsf{i} = \lambda y \colon \mathsf{OM}.\, \mathsf{getint}(y, \mathsf{stack}(\mathsf{ml} = x.\mathsf{stacktop}, \mathsf{cl} = 0)), \\
\quad \mathsf{i\_becomes} = \lambda v \colon \mathsf{int}.\, \lambda y \colon \mathsf{OM}.\, \mathsf{putint}(y, \mathsf{stack}(\mathsf{ml} = x.\mathsf{stacktop}, \mathsf{cl} = 0), v), \\
\;\; \mathsf{IN} \\
\quad [\![\, \texttt{i = 5} \,]\!]\ ; \\
\quad [\![\, \ldots \,]\!]\,) \cdot x
\end{array}
$$

The interpretation $[\![\, \texttt{i = 5} \,]\!]$ of the assignment uses the local function i_becomes.

Again, attributing appropriate positions in cells to local variables is done by the LOOP compiler, and is not a concern for the user. Parameters and return variables are treated similarly.

4.6 Object creation

Creating a new object takes place on the heap of our memory model. Specifically, creating an object of class `C` (via Java's instance creation expression `new C()`) involves the following steps in a memory model $x \colon \mathsf{OM}$.

1. An EmptyObjectCell is stored in the heap memory x.heapmem at location x.heaptop, providing the default values for the fields of `C` (see [6, §§ 4.5.4]) and x.heaptop is incremented by 1.
2. If `C` has static fields (or static initialiser code), then static initialisation is done next, see [6, §§ 12.4], at the static location $s_{\mathtt{C}}$ of `C`—determined by the LOOP compiler. This will make the Boolean $(x.\mathsf{staticmem} \cdot s_{\mathtt{C}}).\mathsf{initialised}$ true—preventing future static initialisation for this class. Static initialisation starts from the top-most superclass of `C`, *i.e.* from `Object`. A subtle matter which is not clearly determined in the Java Language Specification, as

pointed out in [4], is whether static initialisation should also be performed for superinterfaces. In our model it does not happen.
3. Non-static initialisation code of the superclasses of `C` is invoked via appropriate constructors, starting from the top-most superclass `Object`, see [6, §§ 15.8 and 12.5].
4. The relevant constructor plus initialisation code from `C` is executed.
5. A reference to the newly created object at heaptop is returned.

5 Statements and expressions

Statements and expressions in Java may either hang (*i.e.* not terminate at all), terminate normally, or terminate abruptly. Expressions can only terminate abruptly because of an exception (*e.g.* through division by 0), but statements may also terminate abnormally because of a `return`, `break` or `continue` (the latter two with or without label). A `break` statement with a label is used to exit a surrounding (possibly nested) block with the same label. An unlabeled `break` is used to exit the innermost `switch`, `for`, `while` or `do` statement. Within loop statements (`while`, `do` and `for`) a `continue` statement can occur. The effect is that control skips the rest of the loop's body and starts re-evaluating the (update statement, in a `for` loop, and) Boolean expression which controls the loop. A `continue` statement can be labeled, so that the `continue` is applied to the correspondingly labeled loop, and not to the innermost one. More details about the rôle of these abnormalities can be found in [9].

All these options are captured in appropriate datatypes. First, abnormal termination leads to the following types.

– TYPE THEORY

$$\begin{array}{l}\mathsf{StatAbn} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \{\, \mathsf{excp} \colon [\, \mathsf{es} \colon \mathsf{OM}, \mathsf{ex} \colon \mathsf{RefType}\,] \\ \quad \;\mid \mathsf{rtrn} \colon \mathsf{OM} \\ \quad \;\mid \mathsf{break} \colon [\, \mathsf{bs} \colon \mathsf{OM}, \mathsf{blab} \colon \mathsf{lift}[\mathsf{string}]\,] \\ \quad \;\mid \mathsf{cont} \colon [\, \mathsf{cs} \colon \mathsf{OM}, \mathsf{clab} \colon \mathsf{lift}[\mathsf{string}]\,]\,\}\end{array}$$

$$\begin{array}{l}\mathsf{ExprAbn} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad [\, \mathsf{es} \colon \mathsf{OM}, \mathsf{ex} \colon \mathsf{RefType}\,]\end{array}$$

These types are used to define the result types of statements and expressions:

– TYPE THEORY

$$\begin{array}{l}\mathsf{StatResult} : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \{\, \mathsf{hang} \colon \mathsf{unit} \\ \quad \;\mid \mathsf{norm} \colon \mathsf{OM} \\ \quad \;\mid \mathsf{abnorm} \colon \mathsf{StatAbn}\,\}\end{array}$$

$$\begin{array}{l}\mathsf{ExprResult}[\alpha] : \mathsf{TYPE} \stackrel{\mathrm{def}}{=} \\ \quad \{\, \mathsf{hang} \colon \mathsf{unit} \\ \quad \;\mid \mathsf{norm} \colon [\, \mathsf{ns} \colon \mathsf{OM}, \mathsf{res} \colon \alpha\,] \\ \quad \;\mid \mathsf{abnorm} \colon \mathsf{ExprAbn}\,\}\end{array}$$

A Java statement is then translated as a state transformer function $\mathsf{OM} \to \mathsf{StatResult}$, and a Java expression of type `Out` as a function $\mathsf{OM} \to \mathsf{ExprResult[Out]}$. Thus both statements and expressions are coalgebras. The result of such functions applied to a memory $x\colon \mathsf{OM}$ yields either hang, norm, or abnorm (with appropriate parameters), indicating the sort of outcome. A slightly more abstract version of this representation of statements and expressions is presented in terms of a computational monad in [12]. It gives rise to compact versions of the explicit definitions below.

On the basis of this representation of statements and expressions all language constructs from (sequential) Java are translated. For instance, the composition $s\,;t\colon \mathsf{OM} \to \mathsf{StatResult}$ of two statements $s, t\colon \mathsf{OM} \to \mathsf{StatResult}$ is defined as:

– TYPE THEORY

$$
\begin{array}{rl}
(s\,;t)(x) \stackrel{\mathrm{def}}{=} & \mathsf{CASES}\ s \cdot x\ \mathsf{OF}\ \{ \\
& \mid \mathsf{hang}() \mapsto \mathsf{hang}() \\
& \mid \mathsf{norm}\, y \mapsto t \cdot y \\
& \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\, \}
\end{array}
$$

And Java's two conjunction operations `&` and `&&` are defined type-theoretically on $e, d\colon \mathsf{OM} \to \mathsf{ExprResult[bool]}$ as:

– TYPE THEORY

$$
\begin{array}{rl}
(e\ \&\ d) \cdot x \stackrel{\mathrm{def}}{=} & \mathsf{CASES}\ e \cdot x\ \mathsf{OF}\ \{ \\
& \mid \mathsf{hang}() \mapsto \mathsf{hang}() \\
& \mid \mathsf{norm}\, y \mapsto\ \mathsf{CASES}\ d \cdot (y.\mathsf{ns})\ \mathsf{OF}\ \{ \\
& \qquad\qquad \mid \mathsf{hang}() \mapsto \mathsf{hang}() \\
& \qquad\qquad \mid \mathsf{norm}\, z \mapsto \mathsf{norm}(\mathsf{ns} = z.\mathsf{ns}, \mathsf{res} = (y.\mathsf{res} \wedge z.\mathsf{res})) \\
& \qquad\qquad \mid \mathsf{abnorm}\, b \mapsto \mathsf{abnorm}\, b\, \} \\
& \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\, \} \\
(e\ \&\&\ d) \cdot x \stackrel{\mathrm{def}}{=} & \mathsf{CASES}\ e \cdot x\ \mathsf{OF}\ \{ \\
& \mid \mathsf{hang}() \mapsto \mathsf{hang}() \\
& \mid \mathsf{norm}\, y \mapsto\ \mathsf{IF}\ \neg(y.\mathsf{res}) \\
& \qquad\qquad \mathsf{THEN}\ \mathsf{norm}\, y \\
& \qquad\qquad \mathsf{ELSE}\ d \cdot (y.\mathsf{ns}) \\
& \mid \mathsf{abnorm}\, a \mapsto \mathsf{abnorm}\, a\, \}
\end{array}
$$

Notice how the second argument d need not be evaluated for the 'conditional and' operator `&&`, and also how side-effects are passed on via the states $y.\mathsf{ns}$ and $z.\mathsf{ns}$.

Another example, combining expressions and memory access is the the $\mathsf{array\_access}$ function used for translation of indexing an array. It is used as:

$$
[\![\, \mathtt{a[i]}\,]\!] \stackrel{\mathrm{def}}{=} \mathsf{array\_access}(\mathsf{gettyp}, [\![\, \mathtt{a}\,]\!], [\![\, \mathtt{i}\,]\!])
$$

assuming that `a[i]` is not the left hand side of an assignment. The function gettyp is determined by the type of `a`. For example, if `a` is an integer array of type `int[]`, then gettyp = getint. And if `a` is a 2-dimensional array of, say Booleans, then gettyp = getref.

The Java evaluation strategy prescribes that first the array expression, and then the index expression must be evaluated. Subsequently it must be checked first if the array reference is non-null, and then if the (evaluated) index is non-negative and below the length of the array. Only then the memory can be accessed. See [6, §§ 15.12.1 and §§ 15.12.2]. Figure 2 describes array indexing in our setting. It omits the details of how exceptions are thrown.

– TYPE THEORY

```
array_access(gettyp, a, i): OM ⟶ ExprResult[typ]
 def
  =  λx: OM. CASES a · x OF {
                 | hang() ↦ hang()
                 | norm y ↦
                     CASES i · (y.ns) OF {
                        | hang() ↦ hang()
                        | norm z ↦
                            CASES y.res OF {
                               | null() ↦ [[ new NullPointerException() ]]
                               | ref r ↦
                                   CASES r.dimlen OF {
                                      | bot() ↦ hang() // should not happen
                                      | up p ↦
                                          IF z.res < 0 ∨ z.res ≥ p.len
                                          THEN [[ new ArrayIndexOutOf-
                                                       BoundsException() ]]
                                          ELSE norm(ns = z.ns, res =
                                             gettyp(z.ns, heap(ml = r.objpos,
                                                               cl = z.res))) } }
                        | abnorm c ↦ abnorm c }
                 | abnorm b ↦ abnorm b }
```

Fig. 2. Accessing array a : OM → ExprResult[RefType] at index i : OM → ExprResult[int]

Notice that arrays, like objects, are stored on the heap. All translated non-null array references have a non-bottom dimlen field by construction, so in the case indicated as "should not happen" we choose to use hang() as output. We could also have thrown some non-standard exception. There is a similar function array_assign which is used for assigning a value at a particular index in an array. And there is also a function for array creation. It sets up an appropriately linked number of (empty) memory cells, depending on the dimension and lengths of the array that is being created. Space restrictions prevent us from discussing these functions in detail.

6 Examples

We now present three examples of small Java programs from [3], involving aspects that we have discussed above. Hopefully, the reader will appreciate the semantic intricacies.

6.1 Side-effects and Boolean logic

Consider the following Java class.

– JAVA

```
class Logic {
  boolean b, r1, r2;      // r1, r2 will store the result
  boolean f() { b = !b; return b; }
  void m() { b = true;
             r1 = f() || !f();
             r2 = !f() && f(); }
}
```

This example shows that side-effects can disrupt the expected behaviour of disjunction ($\varphi \vee \neg\varphi = \mathsf{true}$) and conjunction ($\neg\varphi \wedge \varphi = \mathsf{false}$): after running method `m()`, the field `r1` will be false, and `r2` will be true. This comes out in our model because of the propagation of side-effects as described in Section 5.

6.2 Logically inconsistent initialisations

Next we consider two mutually recursive classes:

– JAVA

```
class Yes {
  boolean r1, r2, r3;  // for storing the results
  static boolean y = No.n;
  static void m() { r1 = y;
                    r2 = No.n;
                    r3 = No.m(); }
}
class No {
  static boolean n = !Yes.y;
  static boolean m() { return n; }
}
```

From a logical perspective the initialisation is problematic: `Yes.y = No.n` and `No.n = !Yes.y` leading to `Yes.y = !Yes.y`, using substitution[7]. The right view here is of course the operational one. According to the steps in Subsection 4.6, it is clear what will happen: running `Yes.m()`, when both `Yes` and `No` have not been initialised yet, will start the static initialisation of `Yes` by assigning default values to its fields. The static initialisation code `y = No.n` will then start the static initialisation of `No`. The assignment `n = !Yes.y` will use the default value `false` for `y`, so that `n` becomes `true`. Subsequently `y` will also become `true` through the assignment `y = No.n`. Hence `r1`, `r2` and `r3` will all become `true`.

6.3 Order of initialisations

Again we consider two simple Java classes:

– JAVA

```
class Master {
  boolean r1, r2, r3, r4;
  void m() { r1 = Slave.a;
             r2 = Slave.b;
             r3 = Slave.s.a;
             r4 = Slave.s.b; }
}
class Slave {
  static Slave s = new Slave();
  static boolean b = !s.a;
  static boolean a = true;
}
```

The first thing to note is that the initialisation `Slave s = new Slave()` in class `Slave` looks like it will lead to an infinite series of initialisations. But since the field `s` is static, this is a static initialisation that will be done only once. Thus, running `m()` in `Master` will lead to an unproblematic initialisation of `Slave`: first all fields are set to their default values; then the initialisation of `s` will start the initialisation of `b` and `a`. When the value for `b` is computed, `a` still has its default value `false`, so that `b` becomes `true`; only then, `a` becomes `true`. Hence, running `m()`, when `Slave` is not initialised yet, will make all the `r`'s `true`.

7 Verification

Having seen the three examples of Java programs in the previous section, we proceed to discuss how we actually verify (with PVS and Isabelle) that the result variables `r` have the values as indicated. Of course, we cannot go into all details, and therefore we only try to give the reader an impression of what

[7] But from this perspective also an assignment `i = i+1` is problematic!

is going on. We shall present two "user statements" (as in Figure 1), for PVS and Isabelle, and give some further information about the proofs—which in all cases proceed entirely by automatic rewriting[8]. The first example is skipped here because it does not involve significant memory access.

7.1 Verification in PVS

We shall concentrate on the Yes-No example from Subsection 6.2. Running the LOOP tool on these classes will produce several PVS theories, forming the basis of the verification. They are typechecked, together with the semantic prelude containing the definitions for the object memory and for the representations of statements and expressions. Then they are imported into a theory which contains the following user statement.

— PVS

```
IMPORTING ...  % code generated by the LOOP tool is loaded

Yes_lem : LEMMA
  FORALL(p : MemLoc?, x : OM?,
         c : [MemLoc? -> [OM? -> Yes?IFace[OM?]]]) :
    YesAssert?(p)(c(p))
      AND
    p < heap?top(x)
      AND
    NOT Yes_is_Initialized?(x)
      AND
    NOT No_is_Initialized?(x)
        IMPLIES
    norm??(m?(c(p))(x))   % m from Yes terminates normally
      AND
    LET y = ns?(m?(c(p))(x)) IN   % y is 'normal' state after m
      r1(c(p))(y) = TRUE
        AND
      r2(c(p))(y) = TRUE
        AND
      r3(c(p))(y) = TRUE
```

We shall explain some aspects. First of all, there are many question marks '?' in this fragment of PVS code. They are there because they cannot occur in Java identifiers, and so by using them in definitions which are important for the translation to PVS, name clashes are prevented. Further, the statement quantifies first over a memory location p, which is to be understood as the

[8] Proofs only by automatic rewriting are not possible for more complicated programs with `for` and `while` loops. To handle these we use a special version of Hoare logic, tailored to Java, see [9].

location of an arbitrary object, which is required to be in the used part of the heap memory, below the heaptop. The quantification also involves an object memory `x`, and a "coalgebra" `c` incorporating all methods from the class `Yes` in a single function. Notice that this coalgebra (or class) is parametrised by memory locations, so it can act on an arbitrary (heap) location. This coalgebra is required to satisfy a predicate `YesAssert?` ensuring that fields of `c` are bound to appropriate memory positions, and methods and constructors of `c` to their bodies.

Under the additional assumptions that classes `Yes` and `No` have not been initialised yet in state `x`, the statement says that method `m` of coalgebra/class `c` at `p` terminates normally, and that in its result state `y`, all the fields `r` of coalgebra/class `c` are `true`.

The proof in PVS of this statement can be done entirely by automatic rewriting. This involves mimicking the computation on the memory model described in Section 4. The actual PVS proof consists of the following sequence of three proof commands.

– PVS

```
(LOAD-CLASSES ("java_lang_Object" "Yes" "No"))
  (LOAD-PRELUDE) (REDUCE)
```

The first one loads the (automatically generated) rewrite rules for the relevant classes `Yes`, `No` and for the root class `Object`. The next command loads the rewrite rules from the semantic prelude (see Figure 1) into the prover. Finally, the `(REDUCE)` command tells PVS to go off and do any simplification it can apply. It will result in a QED, without further user interaction[9].

7.2 Verification in Isabelle/HOL

Here we concentrate on the Master-Slave example from Subsection 6.3. We proceed in a similar way as with the PVS verification. Thus we run the LOOP tool on the Java classes to generate several Isabelle theories, which are typechecked together with the semantic prelude. This semantic prelude contains similar definitions as the semantic prelude in PVS, but of course it is adapted to the specification language of Isabelle.

Subsequently we prove the following user statement.

[9] This QED involves 177 rewrite steps, which, on a reasonably fast machine according to current standards (a 500Mhz Pentium III with 256M RAM) takes about 40 seconds, most of which are needed for loading the rewrite rules.

— ISABELLE

```
(* Code generated by the LOOP tool is loaded *)
Goal "[|MasterAssert p (c p);\
\       p < heap_top x;\
\       ~ Slave_is_Initialized x |] ==>\
\       let y = PreStatResult.ns(m_ (c p) x)\
\       in r1 (c p) y = True &\
\          r2 (c p) y = True &\
\          r3 (c p) y = True &\
\          r4 (c p) y = True";
```

As in the previous example, `p` denotes the memory location, `c` the coalgebra (or class) and `x` the object memory. Again, we assume that the coalgebra `c` satisfies an assertion `MasterAssert`, ensuring that fields and methods of `c` are appropriately bound. Furthermore, we assume that the class `Slave` is not initialised yet. Under these assumptions we prove that the method `m` terminates normally, resulting in a new (normal) state `y`, in which all the `r`-fields of coalgebra/class `c` are `true`.

The proof in Isabelle is done entirely by automatic rewriting. The actual proof command in Isabelle is the following.

— ISABELLE

```
by (asm_full_simp_tac
     (simpset () addsimps (PreludeRewrites @
                           java_lang_ObjectRewrites @
                           SlaveRewrites @
                           MasterRewrites)) 1);
```

This adds the rewrite rules for the classes `Master`, `Slave` and `Object`, and for the semantic prelude to the simplifier, and subsequently uses these to rewrite the goal to `True`, thus proving the goal[10].

8 Conclusion

We have presented an essential ingredient of the semantics of sequential Java, as used for the tool-assisted verification of Java programs within the LOOP project, namely the object memory used for storing the contents of fields of Java objects in essentially untyped cells. The effectiveness of this formalisation has been demonstrated in the verification of several short, but non-trivial, Java programs. A promising application of this approach, based on the memory model

[10] Isabelle tries to apply over 18,000 rewriting steps, of which only about 800 succeed (including rewrites in conditions of conditional rewrites). This takes approximately 194 seconds on the same 500 Mhz Pentium III with 256M RAM that was used for the PVS verification.

in this paper, is verification of class libraries (like the invariant proof for the `Vector` class in [10]), especially when these libraries are annotated (using the annotation language JML [15] for Java). We are currently developing this approach for the (relatively simple) JavaCard [1] API classes, see [19]. Up-to-date information about the LOOP project is available from the LOOP webpage [20].

References

1. JavaCard API 2.1. `http://java.sun.com/products/javacard/htmldoc/`.
2. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 2nd edition, 1997.
3. J. Bergstra and M. Loots. Empirical semantics for object-oriented programs. Artificial Intelligence Preprint Series nr. 007, Dep. Philosophy, Utrecht Univ., 1999.
4. E. Börger and W. Schulte. Initialization problems in Java. *Software—Concepts and Tools*, 20(4), 1999.
5. J.A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, Cambridge, MA, 1996.
6. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
7. D. Griffioen and M. Huisman. A comparison of PVS and Isabelle/HOL. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics*, number 1479 in Lect. Notes Comp. Sci., pages 123–142. Springer, Berlin, 1998.
8. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. Techn. Rep. CSI-R0004, Comput. Sci. Inst., Univ. of Nijmegen, 2000.
9. M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 284–303. Springer, Berlin, 2000.
10. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. (An earlier version appeared in: B. Jacobs, G.T. Leavens, P. Müller, and A. Poetzsch-Heffter (eds.), Formal Techniques for Java Programs. Proceedings of the ECOOP'99 Workshop. Technical Report 251, Fernuniversität Hagen, 1999, p.37-44), 2000.
11. B. Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, pages 83–103. Kluwer Acad. Publ., 1996.
12. B. Jacobs and E. Poll. A monad for basic Java semantics. In *Algebraic Methodology and Software Technology*, Lect. Notes Comp. Sci. Springer, Berlin, 2000.
13. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
14. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about classes in Java (preliminary report). In *Object-Oriented Programming, Systems, Languages and Applications*, pages 329–340. ACM Press, 1998.
15. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/ leavens/JML.html`), 1999.
16. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.

17. L.C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and computer science*, pages 361–386. Academic Press, London, 1990. The APIC series, vol. 31.
18. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S.D. Swierstra, editor, *Programming Languages and Systems*, Lect. Notes Comp. Sci., pages 162–176. Springer, Berlin, 1999.
19. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. Techn. Rep. CSI-R0005, Comput. Sci. Inst., Univ. of Nijmegen, 2000.
20. Loop Project. `http://www.cs.kun.nl/~bart/LOOP/`.